Local Thresholding on Distributed Hash Tables

Abstract

We present a binary routing tree protocol for distributed hash table overlays. Using this protocol each peer can independently route messages to its parent and two descendants on the fly without any maintenance, global context, and synchronization. The protocol is then extended to support tree change notification with similar efficiency. The resulting tree is almost perfectly dense and balanced, and has $O(1)$ stretch if the distributed hash table is symmetric Chord. We use the tree routing protocol to overcome the main impediment for implementation of local thresholding algorithms in peer-to-peer systems – their requirement for cycle free routing. Direct comparison of a gossip-based algorithm and a corresponding local thresholding algorithm on a majority voting problem reveals that the latter obtains superior accuracy using a fraction of the communication overhead.

Keywords: Distributed Hash Table, Local Thresholding Algorithms, Binary Tree Routing, Gossip Based Algorithms, In-Network Computation, Chord, Symmetric Chord

1. Introduction

In a world of millions of wired devices, in-network computation algorithms provide an intriguing alternative to centralization. Where distributed data is abundant and bandwidth is limited or costly, some applications can only be implemented distributively. Where adverse manipulation and control are a concern, distributed architecture is often preferred over a centralized agent. Finally, scaling an algorithm to the millions of peers often teaches important lessons on asynchrony, speculative execution, and the containment of partial failure, which prove important to more mundane environments such as grid systems.

Algorithms for distributed computation in peer-to-peer systems fall into several categories. Of these, gossip algorithms are possibly the most popular and certainly the most extensively studied [1, 2, 3, 4, 5, 6, 7, 8, 9]. Local thresholding algorithms [10, 11, 12, 13, 14] are comparable to gossip based algorithms because both address similar problems and similarly provide a proof for convergence. Local thresholding algorithms are considered by far more communication efficient than gossip based algorithms. However, they pose far stricter requirements to the underlying routing protocol. A gossip based algorithm basically requires an efficient way in which information can be propagated to random destinations. In contrast, all known local thresholding algorithms require cycle free routing. Often, work on local thresholding algorithms advocates that a routing tree be induced in preprocessing. However, the non-trivial complexity of inducing and maintaining the tree in a dynamic network has so far rendered local algorithms impractical.

This work considers the problem of computation in distributed hash-table (DHT) overlays – the de-facto standard architecture in peer-to-peer networks. Gossip algorithms can easily be implemented on a DHT: If each peer sends messages to a random peer from its finger table then in $O(\log N)$ messages this information will arrive to a random peer. Local thresholding can be implemented in a DHT using one of the existing tree routing protocols. However, existing tree routing protocols [15, 16, 17] are ill-fit for a local thresholding algorithm. Because these protocols were developed mainly to reduce message redundancy in broadcast or convergecast they operate in a top-down or bottom-up manner. Thus, a peer cannot send messages to its tree neighbors without the involvement of either the root (in a top-down protocol) or its entire subtree (in a bottom-up).

This paper makes two main contributions to the state-of-the-art: First, it presents new binary tree routing and change notification protocols for DHT overlays. This tree routing protocol is local and can be used for multi-way communication over the tree, including broadcast and convergecast. The effect of peer joining or leaving is also local and can be detected and notified using no more than six messages that are routed on the tree. The Enabled by the binary tree routing protocol, the second contribution of this paper is a direct comparison of local thresholding and gossip based algorithms. Our experiments show that regardless of system size or properties of the data, local thresholding vastly outperforms gossip. The results are so one sided that they call into question the continued relevance of gossip algorithms to computation in DHT overlays.

The rest of this paper is organized as follows: The next section describes the binary tree routing protocol and the change notification protocol. Section 3 details the implementation of the two majority voting algorithms. Experiments are described in Section 4 and related work in Section 5. Finally, Section 6 draws conclusions and poses some further research problems.

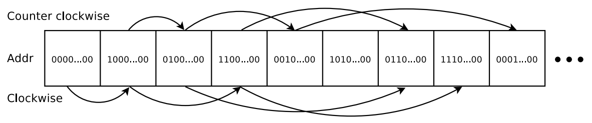
2. Local Binary Tree Routing

The basic idea of the binary tree routing protocol is to define a mapping of peers to a subset of the nodes of a full binary tree. The binary tree can be defined in terms of a one-to-one mapping of d-long binary strings, namely addresses, to tree nodes. Then we define which peer is mapped to which address.

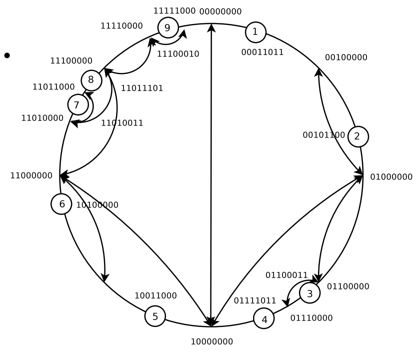
Consider a binary tree whose root is the all zero address. Any address other than that of the root, we divide into three parts: An all zero suffix, which might be empty, the rightmost set bit, and a prefix, which might be empty as well. An address is therefore encoded as $p10^k$, where the length of the prefix p is $d-k-1$. We define that the clockwise descendant of the address $p10^k$ is the address $p110^{k-1}$ and the counterclockwise descendant of the address $p10^k$ is $p010^{k-1}$. Addresses ending with a set bit, i.e., $p10^0$, have no descendants. For completeness, we denote 10^{d-1} the clockwise descendant of the root. As can be seen in Figure 2.1a, this mapping is similar, but not identical to, the textbook implementation of a complete binary tree in an array.

Figure 2.1: Binary Tree routing

(a) Tree mapping to the address space.



(b) Binary Tree Routing on DHT



A peer's position in the tree depends on its assigned address space segment. The peer whose address space segment contains the all zero address takes the root position. Any other peer takes a position calculated as follows: Let the address space segment of p_i be $(a_{i-1}, a_i]$. Let p be the (possibly empty), common prefix of a_{i-1} and a_i, such that $a_{i-1} = p0X$ and $a_i = p1Y$. Then p_i takes the position $p10^k$. Note that messages routed to the address which is a peer's position will always be accepted by that peer.

We conveniently denote the position with which p_i is associated as pos_i. We further denote the clockwise descendant of pos_i as $CW\left[pos_i\right]$ and its counterclockwise descendant as $CCW\left[pos_i\right]$. Respectively, if $pos_j = CW\left[pos_i\right]$ or $pos_j = CCW\left[pos_i\right]$, then we denote $pos_i = UP\left[pos_j\right]$. The functions CW, CCW, and UP can be computed for any pos_i using bit manipulations.

The following two lemmas show that if there is more than one peer in the clockwise (or, respectively, the counterclockwise) subtree of a peer's position then one of those peers occupies a position which is a fore-parent of the positions of all other peers.

Lemma 1. *The address space segment associated with the peers whose positions are the subtree below any peer p_i is continuous.*

Proof. See Appendix A. □

Lemma 2. *For any peer p_i whose position is pos_i, one of the peers which occupies positions in the subtree of position $CW\,[pos_i]$ (respectively, $CCW\,[pos_i]$) occupies a position which is a fore-parent of the positions of all of the other peers in that subtree.*

Proof. See Appendix A. □

Lemma 2 can be put into direct use to define a binary tree of peers, rather than of positions: If all peers in the subtree of the address clockwise from p_i's position have positions which are in the subtree of p_{cw}, then we denote p_{cw} the clockwise neighbor of p_i. Likewise, the counterclockwise neighbor of p_i is the peer p_{ccw}, whose position is the fore-parent of all of the positions in the counterclockwise subtree of pos_i that are occupied by peers.

It remains to define how messages can efficiently be routed from a peer to its neighbors on the tree. The pseudocode of a protocol achieving this is detailed in Alg. 1. To deliver messages to the UP neighbor of a peer p_i, they are first addressed to $UP\,[pos_i]$ and then continue being routed to the $UP\,[pos]$ of that address until they reach an address occupied by a peer. Clockwise and counterclockwise messages are first routed to $CW\,[pos_i]$ and $CCW\,[pos_i]$. If they reach an address not occupied by a peer, this is because the destination falls in the address space segment of a peer p_j occupying a different position. A new destination, which is a step down the tree and away from that of pos_j, is thus computed. If the destination address exhausts the address space, the message is dropped.

Forwarding a message again and again until the destination is found or the address space is exhausted is often wasteful and unnecessary. Whenever the address of the destination position falls in the address space of a peer who has a different position and is also a neighbor of the sender, the message can be dropped. This is because the message is doomed to be sent back and forth between the sender and the receiver until eventually being dropped as there is no peer between them to accept it. Fortunately, such communication patterns can easily be avoided if the sender denotes as part of the message header the edge of its address space in the direction in which the message is sent. The recipient can then compare

Algorithm 1 Local Binary Tree Routing

```
On downcall to SEND with message M and direction d:
If d is upward then $dest \leftarrow UP[pos_i]$ and $edge \leftarrow null$
If d is counterclockwise then $dest \leftarrow CCW[pos_i]$ and $edge \leftarrow a_{i-1}$
If d is clockwise then $dest \leftarrow CW[pos_i]$ and $edge \leftarrow a_i$
Make a downcall to SEND with the destination address $dest$ and the message $\langle pos_i, dest, edge, M\rangle$ using the DHT
On upcall DELIVER with the message $\langle origin, dest, edge, M\rangle$:
If $dest = pos_i$ then call $ACCEPT$ with the message $M$ and finish.
If $dest$ is a fore parent of $origin$ then $newdest \leftarrow UP[dest]$ and $newedge \leftarrow null$
Else if $dest$ is in the clockwise subtree of $origin$ then
- If $edge = a_{i-1}$ then finish
- If $origin = pos_i$ then $newdest \leftarrow CW[dest]$ and $newedge \leftarrow a_i$
- Else $newdest \leftarrow CCW[dest]$ and $newedge \leftarrow a_{i-1}$
Else
- If $edge = a_i$ then finish
- - If $origin = pos_i$ then $newdest \leftarrow CCW[dest]$ and $newedge \leftarrow a_{i-1}$
- - Else $newdest \leftarrow CW[dest]$ and $newedge \leftarrow a_i$
- Make a downcall to SEND with the destination $newdest$ and the message $\langle orig, newdest, newedge, M\rangle$
```

that to the edge of its own address space segment. If the edges are the same, the message can be dropped.

Figure 2.1b illustrates this address scheme in a DHT composed of just nine peers and an address space of eight bits. For instance, peer number 5, whose address space segment is $(01110000, 10011000]$, takes the tree position 10000000, and so forth. Messages routed counterclockwise from 9 first reach position 11010000, which is in the address space segment of peer number 7. However, since the position of peer number 7 is 11000000, the message is then bounced clockwise to position 11011000, which is occupied by peer number 8.

2.1. Tree properties

The properties of the tree which are the most important are its expected maximal depth, the expected degree of internal nodes and the expected stretch of a hop. The expected maximal depth is mostly important for broadcast and convergecast applications, in which a message must traverse the full depth of the tree. The expected degree determines the expansion rate, which is central for repeated averaging algorithms such as gossip algorithms. In any algorithm, actual performance is proportional to the stretch – the number of actual messages needed to deliver a message from a tree node to its parent or its descendant.

Lemma 3. *The maximal depth of the tree is* $O(\log N)$ *where* N *is the number of peers.*

Proof. See Appendix A. □

The stretch of the tree is defined in terms of the number of times the tree routing protocol calculates a new destination for an UP message and lets the DHT route the message. Each such message may require up to $\log N$ IP messages. However, since the tree closely follows the finger table logic, symmetric Chord peers will almost always have a direct link to their CW, CCW and UP neighbors. Therefore, the number of IP messages required for every DHT routing in symmetric Chord is $O(1)$. Notice that messages in the CW and CCW direction follow the same path on the tree as the UP message in the opposite direction, and therefore have the same stretch.

Lemma 4. *The expected stretch of the tree is a small constant.*

Proof. See Appendix A. □

2.2. Neighbor change notification

The binary tree routing protocol in Alg. 1 defines neighbor relations logically and is therefore immune to peer dynamics. Whenever peers join or leave the system the protocol simply reflects the change by delivering messages according to the current tree structure. However, some algorithms, including the one in the next section, still require explicit notification when one of the tree neighbors changes.

The neighbor change notification protocol is based on the following property of the binary tree routing protocol: Let p_i be the successor of p_{i-1} and let their positions be pos_i and pos_{i-1} respectively. If p_{i-1} leaves the system then the position of p_i either remains

Algorithm 2 Neighbor Change Notification

```
Definitions: $Pos\left(a,b\right)$ is the position of a peer whose address space segment is $\left(a,b\right]$
On upcall NOTIFY that the predecessor address has changed from $a_{i-2}$ to $a_{i-1}$ or vice-versa:
Compute $pos_{fix} = Pos\left(a_{i-2},a_i\right)$ and $pos_{var} = \begin{cases} Pos\left(a_{i-1},a_i\right) & Pos\left(a_{i-2},a_{i-1}\right) = pos_{fix} \\ Pos\left(a_{i-2},a_{i-1}\right) & Pos\left(a_{i-1},a_i\right) = pos_{fix} \end{cases}$
Send the message $\langle ALERT, pos_{fix}\rangle$ in direction UP, CW and CCW from $pos_{fix}$ using binary tree routing.
Send the message $\langle ALERT, pos_{vol}\rangle$ in direction UP, CW and CCW from $pos_{vol}$ using binary tree routing.
On upcall ACCEPT with the message $\langle ALERT, pos\rangle$:
If $pos$ is a fore-parent of $pos_i$ then $dir \leftarrow upward$
Else if $pos$ is in the clockwise subtree of $pos_i$ then $dir \leftarrow clockwise$
Else $dir \leftarrow counterclockwise$
Notify the application of a possible change of the neighbor in direction $dir$
```

pos_i or changes to pos_{i-1}. In the former case, the parent of p_{i-1} becomes the parent of its single direct descendant, if one exists. In the latter, p_{i-1}'s former neighbors become the new neighbors of p_i and the former parent of p_i becomes the parent of p_i's former single direct descendant. The same property can prove that it is sufficient to alert those same five peers when p_{i-1} joins the system.

Lemma 5. *The addition or removal of a peer p_i can only affect the tree connectivity of only five peers which are all tree neighbors of either p_i or its successor p_{i+1}.*

Proof. See Appendix A. □

This property can be used to provide alerts on any single local change in topology. This is because when p_{i-1} leaves or joins the system, the DHT alerts its successor that its address space segment has changed. Once p_i is informed of the change in the address space segment it is able to calculate the positions whose neighbors might have changed. Hence, p_i can route alert messages in all directions from those positions.

3. Majority Voting

Given the infrastructure provided by the DHT overlay and by the binary tree routing and change notification protocols, we next compare representative local thresholding and gossip algorithms. We consider the simplest computation task: a majority vote. However, the two algorithms we choose are good representations of their respective families. We use a variant of the local majority voting algorithm of Wolff and Schuster [10] and compare it to LiMoSense [9], which is a variant of the gossip averaging algorithm of Kempe et al. [1], suitable for dynamic data. Both algorithms were slightly adapted, and are therefore described in the following subsections.

The input for both algorithms is a single bit $x_i \in \{0, 1\}$ at each peer p_i and the computational task is to decide if on average most bits are one or zero. We realistically assume that the input of the peers can change at any moment, and thus that the algorithm never terminates. The output of each peer is an ad-hoc assumption on the majority.

3.1. Local majority voting

In local majority voting, every peer bases its output on the statistics of votes it accepts from its tree neighbors – namely: UP, CW, and CCW. The peer stores for each of those neighbors two counter pairs: $X_{UP,i}$ and $X_{i,UP}$ for the upward direction, $X_{CW,i}$ and $X_{i,CW}$ for the clockwise, and $X_{CCW,i}$ and $X_{i,CCW}$ for the counterclockwise direction. Each of those counter pairs counts votes and the number of those votes which are of one. The counter pair $X_{v,i}$ records the latest message received from direction v, and $X_{i,v}$ the latest message sent to direction v. They both are initially $(0, 0)$. We conveniently denote $X_{\perp,i} = (x_i, 1)$ for the input of p_i. The knowledge of a peer is defined as the sum of all its inputs $\mathcal{K}_i = \sum_{d \in \{UP,CW,CCW,\perp\}} X_{d,i}$. Whenever, according to its knowledge, the majority is of ones, $\left(1, -\frac{1}{2}\right)^t \mathcal{K}_i \geq 0$, the peer outputs one. Otherwise, it outputs zero.

To decide when and which messages it must send, the peer computes for every direction $d \in \{UP, CW, CCW\}$ the agreement $\mathcal{A}_{i,d} = X_{d,i} + X_{i,d}$. A violation occurs when for a direction $v \in \{UP, CW, CCW\}$ the sign of the agreement disagrees with the sign of the difference between the knowledge and the agreement: $\left(1, -\frac{1}{2}\right)^t \mathcal{A}_{i,v} \geq 0$ when $\left(1, -\frac{1}{2}\right)^t (\mathcal{K}_i - \mathcal{A}_{i,v}) < 0$ or $\left(1, -\frac{1}{2}\right)^t \mathcal{A}_{i,v} < 0$ when $\left(1, -\frac{1}{2}\right)^t (\mathcal{K}_i - \mathcal{A}_{i,v}) > 0$. Such violations can be triggered by initialization, by a change of the peer's vote, or by an incoming message which changes one of the $X_{d,i}$.

To resolve a violation triggered by the agreement with a neighbor in direction v, a peer can send a message containing information on all of the votes received from neighbors in other directions. This is done by computing $X_{i,v} \leftarrow \mathcal{K}_i - X_{v,i}$ and sending $X_{i,v}$ to the neighbor in the direction v. Notice that after this message is sent, $\mathcal{A}_{i,d} = \mathcal{K}_i$, which resolves the violation.

When a neighbor in direction v changes, the pairs $X_{i,v}$ and $X_{v,i}$ no longer reflect messages sent to or received from the current neighbor. Therefore, when a peer receives an alert of a change in direction v, it sets $X_{v,i}$ to $(0, 0)$ and sends a message to that direction, which sets $\mathcal{A}_{i,v}$ once more to $\mathcal{K}_i$. The change detection protocol alerts the new neighbor as well. So the new neighbor will sending a message which reflects its own knowledge. Once both peers send and accept those messages, $\mathcal{A}_{i,v}$ is again equal to $\mathcal{A}_{v,i}$ and reflects an agreement between p_i and its new neighbor.

Note that if p_i does not have a neighbor in direction v, then $X_{v,i}$ remains zero and does not affect $\mathcal{K}_i$ or the result. Messages sent by p_i in direction v would be dropped by the binary tree routing protocol, but this would not be indicated to p_i. We prefer wasting those messages to complicating the protocol with NACK messages. Additionally, note that to support the possibility of out of order message delivery, a sequential number is attached to each outgoing message and a message is dropped when it arrives after a message which was sent subsequently.

Algorithm 3 DHT Local Majority Voting

```
**Input of peer** $p_i$**:** A vote $x_i \in \{0,1\}$
**Data structure of** $p_i$**:**
$X_{\perp,i}$ initializes to $(x_i, 1)$; $X_{UP,i}$, $X_{i,UP}$,$X_{CW,i}$, $X_{i,CW}$,$X_{CCW,i}$, $X_{i,CCW}$, all initialized to
$(0,0)$, $seq$, $last_{UP}$, $last_{CW}$, $last_{CCW}$ all initialized to 0.
**Output of peer** $p_i$**:** One if $\left(1, -\frac{1}{2}\right)^t \mathcal{K}_i \geq 0$ zero otherwise.
**On change of** $x_i$**:** Set $X_{\perp,i} = (x_i, 1)$ and call test()
**On an upcall to ACCEPT with a message** $\langle X, seq \rangle$ **from** $Pos_j$**:**
Let $v \in \{UP, CW, CCW\}$ be the direction of $Pos_j$ from $Pos_i$.
If $seq > last_v$ then set $X_{v,i} \leftarrow X$, $last_v \leftarrow seq$, and call test()
**On an upcall to ALERT with direction** $v$**:** Set $X_{v,i} \leftarrow (0,0)$ and call Send$(v)$
**Procedure test():**
For $v \in \{UP, CW, CCW\}$, if $\left(1, -\frac{1}{2}\right)^t \mathcal{A}_{i,v} \geq 0$ and $\left(1, -\frac{1}{2}\right)^t (\mathcal{K}_i - \mathcal{A}_{i,v}) < 0$ or
$\left(1, -\frac{1}{2}\right)^t \mathcal{A}_{i,v} \geq 0$ and $\left(1, -\frac{1}{2}\right)^t (\mathcal{K}_i - \mathcal{A}_{i,v}) < 0$ then call Send$(v)$
**Procedure Send**$(v)$**:**
Let $X_{i,v} \leftarrow \mathcal{K}_i - X_{v,i}$, $seq \leftarrow seq + 1$
Send a message $\langle X_{i,v}, seq \rangle$ in direction $v$ using binary tree routing.
```

3.2. Gossip majority voting

The gossip algorithm we use is a variant of LiMoSense [9]. To simplify the description and the experiments, we use the failure free version, which does not handle joining and leaving of peers, or unreliable messaging. We make one important adjustment to the algorithm: instead of selecting the destination uniformly at random we select uniformly from among the different destinations in the peer's finger table. This is justified because in a DHT, following a random finger $O(\log N)$ times will lead to a uniformly picked random peer using just $O(\log N)$ messages. A second change, which is semantic more than algorithmic, is that the output is quantized to either zero or one, in line with the voting problem. A detailed description of LiMoSense is not included here for lack of space.

4. Experimental validation

We conducted two sets of experiments to validate the usefulness of our algorithms. The first experiment evaluates the performance of the binary tree routing protocol in terms of the efficiency of the tree it induces: the degrees of peers, their depth, and the stretch –

the number of real messages required to send a message from a tree node to its neighbor. The second compares the local majority voting algorithm, which uses binary tree routing as the communication infrastructure, to LiMoSense, which does not. The algorithms are compared in terms of their scalability and response to stationary and non-stationary changes in the data.

We employed a standard peer-to-peer network simulator, peersim [18]. The simulator is efficient enough to simulate up to a million peers in some experiments. We used reliable messaging and random network delays of from one to ten simulation cycles. The objective of the delay is not to approximate wall time but rather to decouple the peers and avoid locked-step behavior. When using a Chord overlay, we use an existing add-on to peersim. When using Symmetric Chord [19], we use our own variant, which initializes finger tables accordingly. All measurements are averaged on ten random experiments, using different random seeds.

4.1. Tree Properties

We investigated two key properties of the tree induced by the binary tree routing protocol: The density, the depth, and the stretch.The depth of the tree nodes is the distance from the root to each of them. The depth is important mostly for applications which use global communication such as broadcast and converge-cast because, for those applications, the depth is proportional to the delay. As can be seen in Figure 4.1a, for a tree of N peers, the first $\log N - 2$ levels tend to be completely full. The largest number of peers are at the $\log N$ level of the tree, and the reminder are at a small additional depth. In none of the experiments we conducted, even with a million peers, was a peer ever at a depth greater than $\log(N) + 6$. We conclude that the tree is extremely well balanced.

The stretch of a routing overlay is the number of actual messages needed to deliver a message from a peer to its tree neighbor. This metric assumes most of the cost of the protocol is associated with application level routing decisions (i.e., finding the correct finger, and so forth) and not with network delays.

Figure 4.1b depicts the percentage of neighbors at any given hop distance. It compares the results for a symmetric Chord network of 10,000 and of 100,000 peers. The results are nearly identical: 85% percent of the peers are one or two hops away from their tree neighbors. These results are then contrasted with a (non-symmetric) Chord network of 10,000 peers. In that network the hop distance to a neighbor is a combination of the hop distance to clockwise neighbors, which is the same as that in symmetric Chord, and the hop distance to a counterclockwise neighbor, which is the same as the distance between any two random Chord peers. When using regular Chord overlay, 75 percent of the tree neighbors are within a hop distance of seven or less. Although not as good, the average stretch is still well below $\log N$.

4.2. Majority Voting on DHT

The second set of experiments compares local majority using the binary tree routing protocol in the context of local majority voting with majority based on gossip. We separate the experimented to between those using static votes and those using stationary vote distributions.

4.2.1. Static data

An experiment with static data emulates a snapshot scenario of peer-to-peer computation. In such scenarios, it is assumed that the input is a distributed sample (i.e., snapshot) taken at very large intervals – large enough for the algorithm to stabilize between every two snapshots. The goal of an algorithm in this state is to stabilize as quickly and using as few messages as possible. We leave out experiments with convergence time because of the difficulty of comparing the runtime of a cycle-driven algorithm to an event driven one, because of space considerations, and because the results of the two algorithms did not differ notably in that respect.

The input of the peers is randomly set with an average μ_{pre}. Once all peers compute the same output of the majority function, the input of some peers is randomly switched and the average is set to μ_{post}. At this point, the algorithm proceeds until all of the peers once more compute the correct result. The number of messages needed to reach this point is reported. Three very distinct cases arise: $\mu_{pre} < \frac{1}{2} < \mu_{post}$, $\mu_{pre} < \mu_{post} < \frac{1}{2}$, and $\mu_{post} < \mu_{pre} < \frac{1}{2}$. Other arrangements of μ_{pre}, of μ_{post}, and of $\frac{1}{2}$ are symmetric because both algorithms have no preference for a majority of ones or of zeros. In the last of the three cases, convergence is instantaneous in both algorithms because no peer ever outputs the wrong majority. We therefore focus on the former two cases and experiment with two main arguments: The scale – number of peers, and the signal – distances of μ_{pre} from μ_{post}, and from $\frac{1}{2}$.

Figure 4.2 depicts the number of messages per peer required for each of the algorithms so that all peers compute the correct majority on networks of 10,000 to 160,000 peers. The experiments in Figure 4.2a depict the first case, with μ_{pre} and μ_{post} varied from 10% vs. 90% through to 40% vs. 60%. The most evident outcome of these experiments is that local majority is by far better than LiMoSense in this metric.

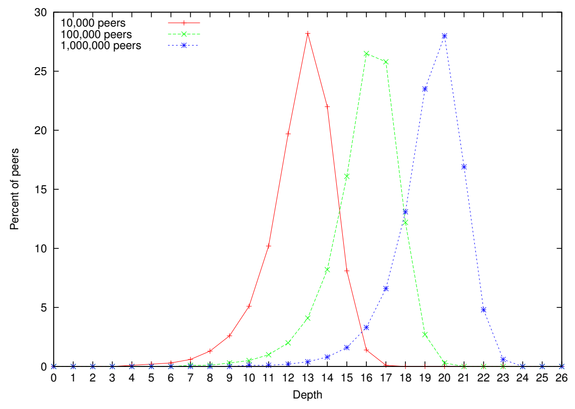
The reason for the difference may be simple: in local majority, it does not take long until only a few peers continue to exchange messages. In LiMoSense, as well as in similar gossip algorithms, peers continue to send messages periodically until the stopping criterion is reached. In this experiment, the stopping criterion is that the last peer has computed the correct result. However, gossip would remain inefficient if other stopping criteria, such as a fixed number of cycles, or a decrease of variance to some degree, are used. It is the data dependency of the local thresholding algorithm which makes the difference.

4.2.2. Stationary data

In an experiment with stationary votes, a number of peers are randomly picked at every given period and their vote is switched, keeping the overall proportion of zero to one votes constant. We denote the fraction of peers whose input changes at each average message

Figure 4.1: Tree depth and stretch

(a) Distribution of peer depth



(b) Number of hops to tree neighbor

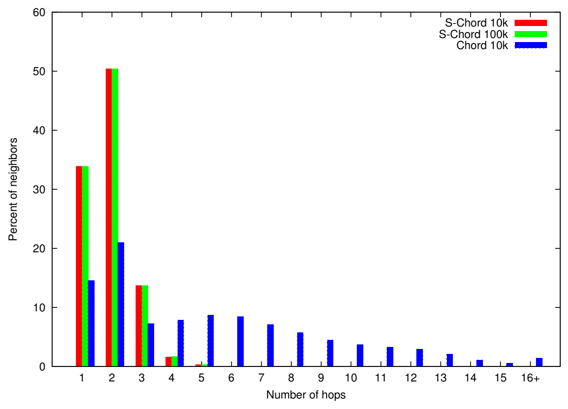
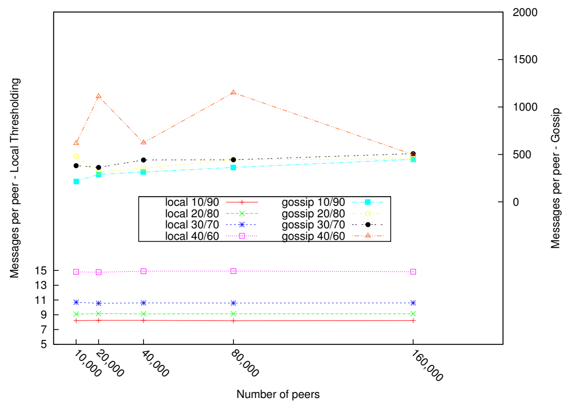
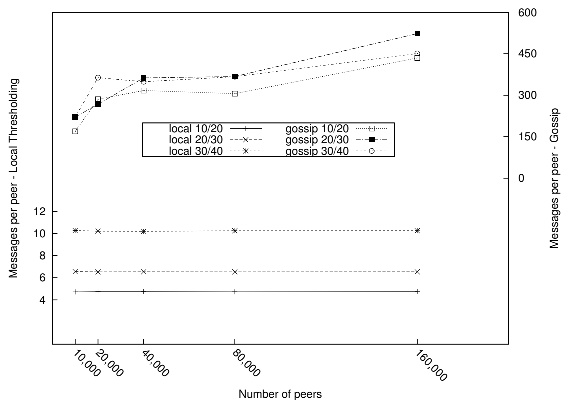


Figure 4.2: Messages until convergence with static data

(a) Output changes



(b) Output remains the same



delay (five simulation cycles) the *noise rate* and measure it in peers per million per cycle (ppm/c).

When inputs constantly change, convergence is impossible and convergence cost becomes meaningless. Instead, it is the proportion of peers which compute the correct outcome (i.e., the average accuracy) that matters, and the ongoing communication costs required to preserve this level of correctness. A second question is how well is the performance preserved when the number of peer in the system grows.

Figures 4.3a and 4.3b depict the accuracy and cost of local majority voting for networks of 10,000 to 160,000 peers and at various noise rates. As can be seen, regardless of the noise rate, both average accuracy and average cost remain constant when the system is scaled-up. Furthermore, even when more than one peer in a thousand changes at every simulator cycle, the accuracy remains above 90%, and fewer than 2% of the peers send a message at every simulator cycle.

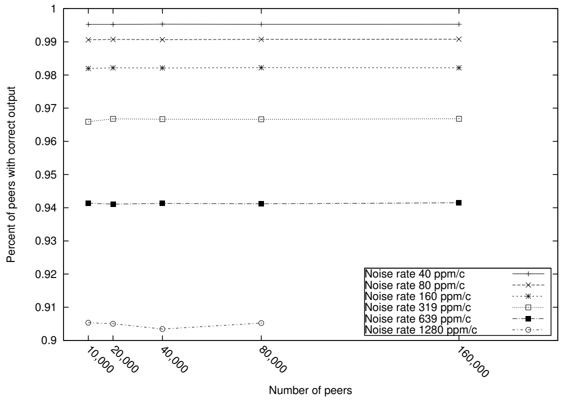
Finally, Figure 4.3c compares the performance of local majority voting is compared to that of LiMoSense. To compare the two algorithms on equal terms, the message overhead of LiMoSense is set to exactly that of local majority voting. Then, LiMoSense is allowed to send from twice that number to 256 times the number of messages local majority voting sends. As can be seen, the utility of LiMoSense does not degrade with scale. However, even when allowed a number of messages which is eight orders of magnitude larger than that of local majority voting, more than twice as many peers err, on average, in LiMoSense. In terms of utility vs. cost, local majority is overwhelmingly superior.

5. Related Work

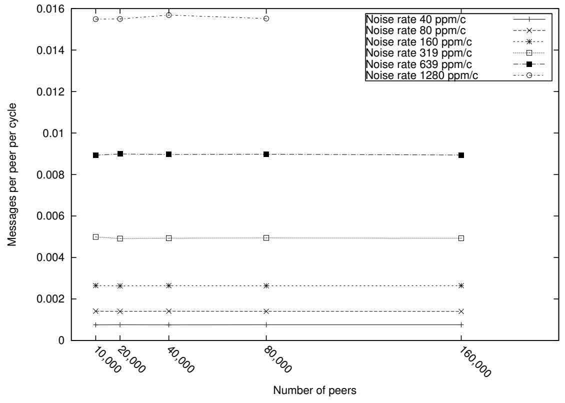
The work described here relates to work in two areas: computation and use of spanning trees in DHT overlays, and computation of majority voting in those and other networks.

Figure 4.3: Scalability of local majority on stationary data

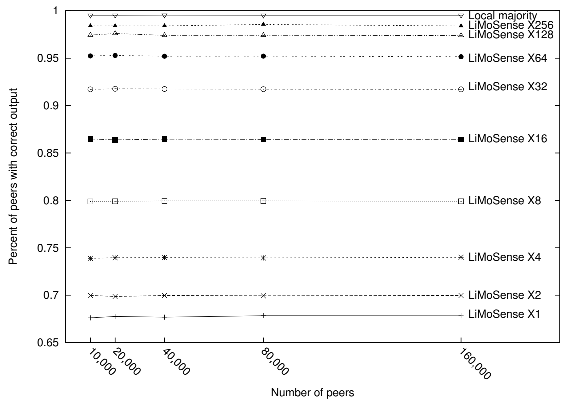
(a) Local majority utility



(b) Local majority cost



(c) Local vs. Gossip Majority Voting



5.1. Spanning trees in DHT overlays

Bottom-up trees are discussed as part of the Scribe system [20], in which peers are organized in groups and the peer whose address is the closest to the groupId is the root. Reverse-path forwarding allows broadcast in Scribe, but a single peer joining a group can alter the parenthood relation.

El-Ansary et al. [15] describe a partition based broadcast tree that does not assure cycle freedom. Huang and Zhang [17] improve on that with a protocol that assures cycle freedom but in which peer degrees vary from zero to $\log N$. Lately Huang and Zhang [16] further improved their protocol with balanced DBT in which the right and left descendants of a peer are, respectively, the next peer and the peer responsible for the middle address of the address space of the parent. Each peer then distributes the broadcast to half the address space of the parent. Balanced DBT offers both a bounded out-degree of two and a stretch that is typically one. The binary tree routing protocol presented here further improves on balanced DBT by removing the need for global partitioning of the address space. Thus, it allows not only broadcast, but also convergecast, or multi-way cycle free communication, which is the way local majority voting uses it.

5.2. Distributed majority voting

The computation of the majority has long been a focal point of algorithms intended for in-network computation. It was the subject of the first local data mining algorithm [10], and is a straightforward reduction of the push-sum gossip based protocol of Kempe et al. [1]. Gossip based algorithms for majority voting were proposed [21, 5]. However, they relate to the problem of limiting the space needed by gossip and do not improve the messaging overhead beyond push-sup or LiMoSense [9].

Birk et al. [22] suggested a local majority voting algorithm for general networks. In that work, each "1" vote spans a tree using the Bellman-Ford algorithm until is either nulled by a "0" vote or it runs against the tree of another "1" vote. The work has several limitations: Trees are data dependent, so they have to be maintained when the data changes. Also, if multiple majority votes are to be taken at once (as often happens in peer-to-peer data mining), then different trees are computed for each vote, and expenses accumulate. Bellman-Ford also incurs significant synchronization overhead between the branches of the tree. In contrast, the local majority voting algorithm described here relies on a binary tree protocol which is data independent and only requires (local) maintenance on (local) topology changes.

6. Conclusions and future work

For almost a decade since gossip based and local thresholding algorithms were first described, the former remain the more practical and the latter the more theoretically efficient. Our binary tree routing protocol begins to bridge the gap. While interesting in itself, the protocol is important because it permits seamless execution of any local thresholding algorithm on a DHT overlay. The two kinds of algorithms can thus be realistically compared. We believe the conclusion of such comparison is beyond doubt: gossip based algorithms are by far inferior to local thresholding algorithms for computation in DHT overlays.

The biggest challenge remaining is computation of local thresholding algorithms on unstructured networks and on networks where communication is noisy and asymmetric. Additionally, we see two interesting challenges in implementing the binary tree protocol for other structured topologies, and generalizing the protocol for trees of greater degree. Such generalization may also serve as a means for controlling communication overhead which, although low, is an artifact rather than an argument of current local thresholding algorithms.

References

[1] D. Kempe, A. Dobra, J. Gehrke, Gossip-Based Computation of Aggregate Information, in: Proceedings of Foundations of Computer Science (FOCS), 2003, pp. 482–

491.

[2] S. Boyd, A. Ghosh, B. Prabhakar, D. Shah, Gossip algorithms: Design, Analysis and Applications, in: Proceedings of INFOCOM 2005, Vol. 3, 2005, pp. 1653–1664.

[3] M. Jelasity, A. Montresor, O. Babaoglu, Gossip-Based Aggregation in Large Dynamic Networks, ACM Transactions on Computer Systems 23 (3) (2005) 219–252. doi:{http://doi.acm.org/10.1145/1082469.1082470}. URL http://doi.acm.org/10.1145/1082469.1082470

[4] M. Cao, D. A. Spielman, E. M. Yeh, Accelerated Gossip Algorithms for Distributed Computation, in: 44th Annual Allerton Conference on Communication, Control, and Computation, 2010.

[5] F. Benezit, P. Thiran, M. Vetterli, The Distributed Multiple Voting Problem, IEEE Journal of Selected Topics in Signal Processing 5 (4) (2011) 791–804. doi:{10.1109/JSTSP.2011.2114326}.

[6] S. Kar, J. M. F. Moura, Gossip and Distributed Kalman Filtering: Weak Consensus under Weak Detectability, IEEE Transactions on Signal Processing 59 (4) (2011) 1766 – 1784.

[7] R. Carli, A. Chiuso, L. Schenato, S. Zampieri, Distributed Kalman Filtering Based on Consensus Strategies , IEEE Journal on Selected Areas in Communications 26 (4) (2008) 622 – 633.

[8] J. B. M. Rabbat, R. Nowak, Robust Decentralized Source Localization via Averaging, in: Proceedings of ICASSP, 2005, pp. 1057–1060.

[9] I. Eyal, I. Keidar, R. Rom, LiMoSense - Live Monitoring in Dynamic Sensor Networks, in: Proceedings of ALGOSENSOR, 2011.

[10] R. Wolff, A. Schuster, Mining Association Rules in Peer-to-Peer Systems, IEEE Transactions on Systems, Man and Cybernetics - Part B 34 (6) (2004) 2426–2438.

[11] P. Luo, H. Xiong, K. , Lü, Z. Shi, Distributed Classification in Peer-to-Peer Networks, in: Proceedings of ACM SIGKDD, 2007, pp. 968–976. `doi:{http://doi.acm.org/10.1145/1281192.1281296}`. URL `http://doi.acm.org/10.1145/1281192.1281296`

[12] R. Wolff, K. Bhaduri, H. Kargupta, Local L2 Thresholding Based Data Mining in Peer-to-Peer Systems, in: Proceedings of SDM, 2006, pp. 430–441.

[13] K. Bhaduri, R. Wolff, C. Giannella, H. Kargupta, Distributed Decision-Tree Induction in Peer-to-Peer Systems, Stat. Anal. Data Min. 1 (2) (2008) 85–103. `doi:{10.1002/sam.v1:2}`. URL `http://dl.acm.org/citation.cfm?id=1388350.1388354`

[14] K. Bhaduri, K. Das, K. Borne, C. Giannella, T. Mahule, H. Kargupta, Scalable, Asynchronous, Distributed Eigen Monitoring of Astronomy Data Streams, Stat. Anal. Data Min. 4 (3) (2011) 336–352. `doi:{http://dx.doi.org/10.1002/sam.10120}`. URL `http://dx.doi.org/10.1002/sam.10120`

[15] S. El-Ansary, L. O. Alima, P. Brand, S. Haridi, Efficient Broadcast in Structured P2P Networks, in: 2nd International Workshop on Peer-to-Peer Systems, 2003, pp. 304–314.

[16] K. Huang, D. Zhang, A Partition-Based Broadcast Algorithm over DHT for Large-Scale Computing Infrastructures, Advances in Grid and Pervasive Computing 5529 (2009) 422–433.

[17] K. Huang, D. Zhang, DHT-Based Lightweight Broadcast Algorithms in Large-Scale Computing Infrastructu
Future Generation Computer Systems 26 (3) (2010) 291–303.
doi:{10.1016/j.future.2009.08.013}.
URL http://dl.acm.org/citation.cfm?id=1663660.1663988

[18] A. Montresor, M. Jelasity, PeerSim: A Scalable P2P Simulator, in: Proc. of the 9th Int. Conference on Peer-to-Peer (P2P), 2009, pp. 99–100.

[19] V. Mesaros, B. Carton, P. V. Roy, S-Chord: Using Symmetry to Improve Lookup Efficiency in Chord., in: Proceedings of PDPTA, 2003, pp. 1752–1760.

[20] M. Castro, P. Druschel, A.-M. Kermarrec, A. Rowstron, SCRIBE: A Large-Scale and Decentralized Application-Level Multicast Infrastructure, IEEE Journal on Selected Areas in Communications 20.

[21] F. Benezit, P. Thiran, M. Vetterli, Interval Consensus: From Quantized Gossip to Voting, in: Proceedings of ICASSP, 2009, pp. 3661–3664.

[22] Y. Birk, L. Liss, A. Schuster, R. Wolff, A Local Algorithm for Ad Hoc Majority Voting via Charge Fusion, in: Proceedings of DISC, Amsterdam , 2004.

[23] C. Penland, The distribution of random segment lengths, Geophysical Research Letters 15 (12) (1988) 1405–1408.

Appendix A. Proofs

Lemma 6. *The address space segment associated with the peers whose positions are the subtree below any peer p_i is continuous.*

Proof. Assume not then p_i cannot be the root because its subtree is all of the peers who, together, are associated with the entire address space. Assume, without loosing generality that p_i is in the position $pos_i = p10^k$. For p_i to have a discontinuous address space there must be at least three peers p_{cw}, p_m, and p_{ccw} such that p_{cw} is clockwise from p_m which is clockwise from p_{ccw} and such that p_{cw} and p_{ccw} are in the subtree of p_i but p_m is not.

Since p_{cw} is in p_i's subtree we know its position pos_{cw} begins with the prefix p and the same goes for the position pos_{ccw} of p_{ccw}. Since the position correspond to an address in the peers address space segment, we know all of the addresses in the address space segment of p_m begin with the prefix p. One of those addresses correspond to p_m's position pos_m which must therefore begin with the prefix p. Whatever that position is, by applying the UP operator to it again and again we will reach pos_i. Hence, p_m is in p_i's subtree, which contradicts the premise. □

Lemma 7. *For any peer p_i whose position is pos_i, one of the peers which occupies positions in the subtree of position $CW\left[pos_i\right]$ (respectively, $CCW\left[pos_i\right]$) occupies a position which is a fore-parent of the positions of all of the other peers in that subtree.*

The lemma holds trivially if there are no peers or just one peer in the subtree. Assuming there is more than one peer in the subtree, let pos_p be the lower common parent position of the positions of all peers in the subtree. If pos_p is occupied by one of those peers, then the lemma is satisfied. Otherwise, pos_p is not occupied by a peer, possible only if it is in the address space segment of a peer p_j which occupies another position pos_j. Since pos_p is the lowest common parent, some of the other peers occupy positions in $CW\left[pos_p\right]$ and some in $CCW\left[pos_p\right]$. This means p_j cannot be equal to p_i, since we know that all the peers are in the subtree of $CW\left[pos_i\right]$ (respectively, the subtree of $CCW\left[pos_i\right]$) . We are left with the conclusion that pos_p is in the address space segment of a peer not in p_i's subtree. However, this is in violation of Lemma 1.

Lemma 8. *The maximal depth of the tree is $O\left(\log N\right)$ where N is the number of peers.*

Proof. The binary tree is fully defined in terms of addresses regardless of the positions actually occupied by peers. The clockwise and the counter-clockwise subtrees of a peer at any address are span equal address spaces. The peers, on the other hand, are randomly and uniformly distributed in the address space. Hence, if the subtree of a peer contains k peers then the number of peers in every subtree is distributed $Bin\left(\frac{1}{2}, k\right)$.

In a binary search tree built from random insertions, the distribution of the number items in every subtree is uniform. It is known that the maximal depth of a random binary search tree is roughly $4.3\log N$. Since the probability that a subtree has more than $\frac{k}{2}+i$ nodes is higher in a random binary search tree than it is in the tree induced by the protocol, the maximal depth of the tree which is induced is expected to be smaller than $4.3\log N$. □

Lemma 9. *The expected stretch of the tree is a small constant.*

Proof. Call the destination address of the first hop the first address, and that of the second hop the second address. Any address between that of the initiator and the first and second addresses must be part of the subtree of either the initiator or the first and the second peer, respectively, because of Lemma 1. If there are more than two hops then the destination of the third hop must be in the same address space segment as the first address. Or else, the first address would be the highest in its address space segment, and thus would be occupied by a peer which would become the parent of the initiator. The same is true for the destination of the forth hop, if there is one, and the second address. We conclude that if a message in the UP direction makes more than two hops then those hops are between two distinct peers – the first and the second one – and that eventually, one of those peers must be the parent of the initiator.

The distance between the first and the second addresses must be larger than the address space segment of the initiator. The distance between the first and the third destinations is at least as large. If the third destination is not the parent's address then the address space segment which includes both the first and the third destinations must be at least three time larger than the address space segment of the initiator. Respectively, if the forth hop is not the last then the address space segment of the second peer must be at least seven times larger than the initiator's address space. In general, if the message hops $k > 2$ times then the address space segment of both the first and the second peer must be at least $2^{k-2} - 1$ larger than that of the initiator.

It is known [23] that the length of uniform random segments is exponentially distributed. Given that the size of an address space segment is c, the probability that the size of the consecutive segments is $c \cdot 2^k$ is the probability of sampling both values from the exponential distribution, $Pr = \left(1 - e^{-c\lambda}\right) e^{-c2^k\lambda}$. For any constant c, this probability decreases double exponentially in k. We conclude that the expected number of hops between a peer and its parent is a constant not much greater than three. □

Lemma 10. *The addition or removal of a peer p_i can only affect the tree connectivity of only five peers which are all tree neighbors of either p_i or its successor p_{i+1}.*

Proof. When p_i is added, the address space segment of its successor p_{i+1} is divided between p_i and p_{i+1}. One of the peers receives the address which previously corresponded with pos_{i+1}. Clearly, if that peer is p_i then the connectivity of the peers which previously where the parent and direct descendants of p_{i+1} changes, since p_i now replaces p_{i+1} as their neighbor. The other peer, be it p_i or p_{i+1}, receives a new position. Call this peer p_{new} and its position pos_{new}. Previous to p_i addition, pos_{new} was not occupied by a peer because the corresponding address was part of the address space of p_{i+1} and that address space included a higher position – pos_{i+1}. The addresses between that corresponding with pos_{new} and that which corresponds to pos_{i+1} are all in the address space of either p_i or p_{i+1}. Those addresses all correspond to positions which are lower than the positions occupied by the two peers. Therefore, p_{new} can have at most one descendant. When a message was previously routed up from that possible descendant, it was routed to pos_{new} and then forwarded further up because pos_{new} was part of an address space belonging to a peer with a different position. Therefore, whichever is the peer which now accept messages sent up from p_{new}, that peer was the previous parent of p_{new} sole possible descendant. Because no other address space segment changes, no other peer changes its position. Since we already enumerated the neighbors of p_i and p_{i+1}, the connectivity of any peer other than those neighbors does not change. $\square$